Multiple Materialized View Selection for XPath Query Rewriting

Nan Tang [†], Jeffrey Xu Yu [†], M. Tamer Özsu [‡], Byron Choi [§], Kam-Fai Wong [†]

[†]*The Chinese University of Hong Kong*
{ntang,yu,kfwong}@se.cuhk.edu.hk

[‡]*University of Waterloo*
tozsu@cs.uwaterloo.ca

[§]*Nanyang Technological University*
kkchoi@ntu.edu.sg

***Abstract*— We study the problem of answering XPATH queries using multiple materialized views. Despite the efforts on answering queries using single materialized view, answering queries using multiple views remains relatively new. We address two important aspects of this problem: multiple-view selection and equivalent multiple-view rewriting. With regards to the first problem, we propose an NFA-based approach (called VFILTER) to filter views that cannot be used to answer a given query. We then present the criterion for multiple view/query answerability. Based on the output of VFILTER, we further propose a heuristic method to identify a minimal view set that can answer a given query. For the problem of multiple-view rewriting, we first refine the materialized fragments of each selected view (like pushing selection), we then join the refined fragments utilizing an encoding scheme. Finally, we extract the result of the query from the materialized fragments of a single view. Experiments show the efficiency of our approach.**

I. INTRODUCTION

As XML has become a universal medium for data exchange over the Internet, efficient XPATH query processing has become the focus of considerable research and development activities. In addition to using structural summaries [1]–[6] and holistic join algorithms [7]–[9] to optimize the evaluation of XPATH queries, answering them using XML views has been studied.

There are two fundamental problems in answering queries using materialized views: (1) View(s)/query answerability: finding view(s) that can answer a query; and (2) Query evaluation on view(s): finding compensating queries (i.e., queries that can be applied to materialized views) to obtain the result of the original query.

These problems have been extensively studied in relational systems [10]. Views are also incorporated in many relational database systems, e.g., ZETA [11], System R [12] and INGRES [13]. However, addressing these problems in XML presents new challenges. For the first problem, finding a containment relationship between a view and a query is coNP-complete [14], [15], even for XPATH queries with a restricted syntax including the child-axis ($/$), the descendant-axis ($//$), wildcards ($*$) and branches ($[\cdots]$). Furthermore, to select view(s) that answers a query when a large number of views are materialized, novel techniques are required to avoid naively comparing each view (or combinations of views) with the query. For the second problem, applying compensating queries on materialized views may require accessing the base data, which may introduce high I/O cost leading to low performance.

Many works address the first problem through finding a containment matching (called homomorphism) from a view to a query (e.g., [16]–[18]) in PTIME, which is sound but incomplete. These approaches can handle a fragment of XPATH queries including $/, //, *$ and $[\cdots]$. Some of these works achieve this complexity by restricting the XPATH expression (e.g., by eliminating "$*$" [17]). Above approaches focus on the case of single view/query answerability, where view selection is irrelevant. Mandhani and Suciu [19] discuss single-view selection when there are a large number of views to consider. Answering queries using multiple views can improve query performance and discover the potential connection between views. However, multiple-view selection is not discussed in previous works.

To tackle the second problem, namely query rewriting using views, most works generate the compensating query by stitching up the query predicates that are not satisfied by the view in homomorphism (e.g., [16], [18]). However, these approaches may require accessing the base data to answer a query. For example, a query $a[.//b//d]//c$ is contained in a view $a[.//b]//c$, but it cannot be answered by the view without accessing the base data, since only the XML fragments for node c (not nodes a and b) of the view are materialized, due to XPATH semantics. Alternative approaches that only utilize the materialized fragments [17], [19] include maximal contained rewriting [17] or equivalent rewriting but with limited cases [19] (e.g. $//b//c$ answers $//b//c//d$ but not $//b[.//d]//c$ or $//a//b//c$). Moreover, these approaches address the problem of rewriting using only a single view, without considering the connection among multiple views.

In this paper, we address these problems in a more general setting, which describes a framework of answering XPATH queries using multiple views (see Figure 1). Here, each view is materialized, i.e., the XML fragments for the answer node are pre-computed and stored. (1) Given a query Q and a view set $\mathcal{V} = \{V_1, \cdots, V_n\}$, we propose a *Nondeterministic*

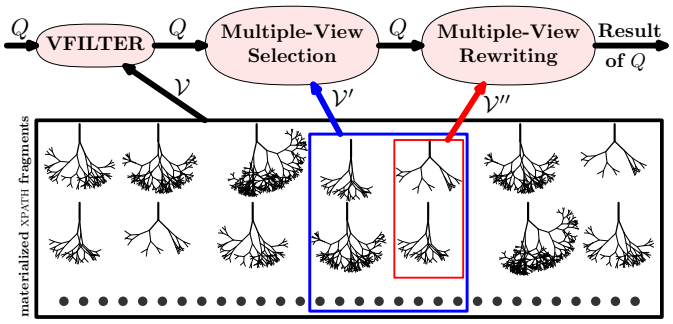

Fig. 1. Framework Overview

Finite Automata (NFA) based approach (called VFILTER) to filter views that cannot be used to answer a query, and obtain a candidate view set $\mathcal{V}'$. (2) We discuss the problem of multiple-view selection. Single-view selection requires at most one scan of each view; for multiple-view selection, however, we need to search all combinations of views (e.g., $\{V_1\}$, $\{V_2\}$, $\{V_1, V_2\} \cdots$) to find the minimum (i.e., smallest) view set that can answer the query. The number of subsets of views is $O(2^n)$. Based on the output of VFILTER, we further propose a heuristic method to identify a minimal view set $\mathcal{V}''$ that answers Q (i.e., $\nexists \mathcal{V}''' \subseteq \mathcal{V}''$ answers Q). (3) We study equivalent multiple-view rewriting. We refine and join the materialized fragments of views in $\mathcal{V}''$ to extract the result of Q.

The main contributions of this paper are the followings:

- We describe a novel NFA-based approach (called VFILTER), which can efficiently filter a large number of views that are irrelevant to answering a given query.
- We present the criterion for multiple view/query answerability. We further propose a heuristic method to efficiently identify a minimal view set to answer a query, based on the output of VFILTER.
- We propose an approach for rewriting a query using multiple views, which uses the materialized view fragments only. Joins between different view fragments are conducted using an encoding scheme without accessing the base data.
- We conduct extensive experiments to show the efficiency of our approaches.

The rest of this paper is organized as follows. We start with an introduction of XML data and queries in Section II. Section III describes an NFA-based approach for view filtering. Section IV discusses multiple-view selection. Section V studies the problem of query rewriting using multiple views. In Section VI, we present our experimental results. We conclude this paper in Section VII.

II. Preliminaries

XML data model. We model XML data as an unordered tree $\mathcal{T}$. Each node x has a label, denoted as $\text{LABEL}(x)$, over a finite alphabet L. The root of $\mathcal{T}$ is denoted as $\text{ROOT}(\mathcal{T})$. An XML tree for `book.xml`[1] is shown in Figure 2, with 34 nodes. Let $\mathsf{L} = \{b, t, a, s, p, f, i\}$, where b, t, a, s, p, f and i are the abbreviations for *book, title, author, section, paragraph, figure* and *image*, respectively. Numeric subscripts are used to distinguish different nodes with the same label.

Encoding schemes. Encoding schemes (e.g., [20]–[23]) are widely used in XML query processing to identify the relationship between XML tree nodes. We adopt `extended Dewey-code` [22] in this paper, and the encoding of each node, in Figure 2, is shown on top of the label. The `extended Dewey-code` of each node can be converted into a *label-path*, i.e., a sequence of node labels from root to a particular node, using a *finite state transducer* (FST) [22]. The FST of the XML tree in Figure 2 is given in Figure 3.

Example 2.1: *Consider the encoding* `0.8.6` *in Figure 2 (for* s_3*). The first label* b *can be derived, since the first number* `0` *in* `0.8.6` *satisfies* `0 mod 1 = 0` *from the input of* FST *in Figure 3. Then label* s *will be derived, since the second number* `8` *has* `8 mod 3 = 2` *on node* `b` *in* FST. *Finally, we derive label* s*, since the last number* `6` *has* `6 mod 4 = 2` *on node* `s` *in* FST. *The label-path of* s_3 *is derived as* $b/s/s$.

With the derived paths from encodings, we can determine whether two nodes satisfy some structural condition without accessing the base data. For example, node t_4 (`0.8.6.0`) and p_3 (`0.8.6.1`) have two common ancestors labeled s, since their common prefix is `0.8.6` that can be derived as $b/s/s$.

XPATH query and tree pattern. The set of XPATH queries studied in this paper is a fragment of XPATH queries with the child-axis ($/$), the descendant-axis ($//$), wildcards ($*$) and branches ($[]$). An XPATH query can be represented as a tree pattern. A *tree pattern* P is an unordered tree. A node n of P has a label, denoted as $\text{LABEL}(n)$, over $\mathsf{L} \cup \{*\}$ where L is the finite alphabet of XML tree; an edge e has a label, denoted as $\text{EDGE}(e)$, from $\{/, //\}$, where $/$ is the *child*-edge and $//$ is the *descendant*-edge. The root of P is denoted as $\text{ROOT}(P)$. The *answer* node specifies the results returned by P, denoted as $\text{RET}(P)$. A tree pattern without branches is called a *path pattern*. We may use "XPATH query" and "tree pattern query" interchangeably in this paper.

We define an *embedding* to be a mapping f from a tree pattern P to an XML tree $\mathcal{T}$ where (1) $\text{LABEL}(f(n)) = \text{LABEL}(n)$ or $\text{LABEL}(n) = *$, for each node n in P; (2) for each edge $e{:}(n_1, n_2)$ in P, if $\text{EDGE}(e){=}/$, $f(n_1)$ is the parent of $f(n_2)$ in $\mathcal{T}$; otherwise $f(n_1)$ is a proper ancestor of $f(n_2)$. For example, there is an embedding from tree pattern $b[a]//t$ to the XML tree in Figure 2, with $f(b) = b_1$, $f(a) = a_1$ and $f(t) = t_2$.

We borrow the definition of *containment* of tree patterns from [14]. For a tree pattern P, $P(\mathcal{D})$ denotes the *boolean* result of evaluating P over an XML database $\mathcal{D}$. We say $P(\mathcal{D})$ is true if there exists an embedding of P in $\mathcal{D}$; it is false otherwise. For two tree patterns P, Q, P is *contained* in Q,

[1] http://www.galaxquery.com/demo/docs/book1.xml

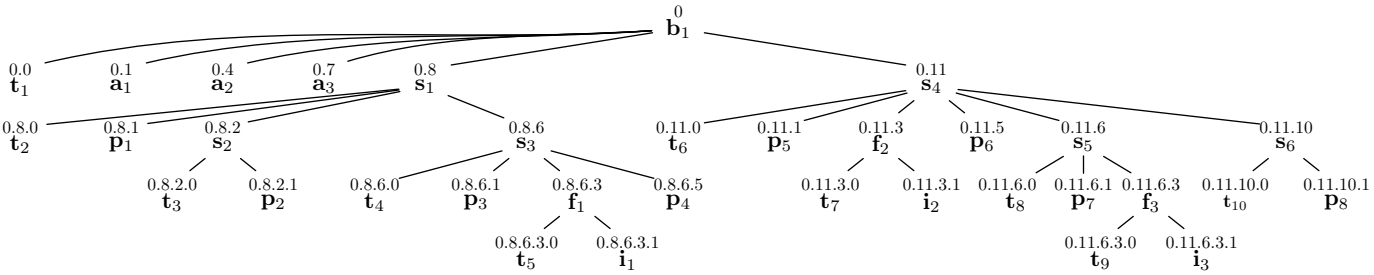

Fig. 2. An XML tree with extended Dewey-code

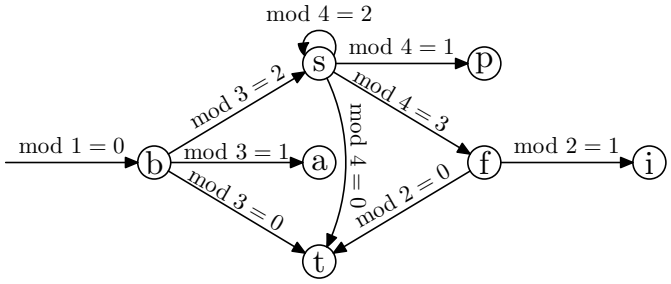

Fig. 3. Finite state transducer of Figure 2

denoted as $P \sqsubseteq Q$, iff $P(\mathcal{D})$ implies $Q(\mathcal{D})$, in every XML database $\mathcal{D}$. We say P and Q are *equivalent*, denoted as $P \equiv Q$, iff $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

Determining tree pattern containment is a coNP-complete problem [14], e.g., by canonical models [15]. A commonly used method for rewriting (in PTIME) is to compute whether there exists a homomorphism (a mapping h) from a tree pattern P to a tree pattern Q (e.g., [18]). Specifically, given two tree patterns P, Q, there is a homomorphism h from P to Q if: (1) $\text{LABEL}(n) = \text{LABEL}(h(n))$ or $\text{LABEL}(n) = *$ for each node n in P. (2) for each edge (n_1, n_2) in P, if $\text{EDGE}(n_1, n_2) = /$ then $\text{EDGE}(h(n_1), h(n_2))=/$ in Q; otherwise $\text{EDGE}(n_1, n_2)=$ $/\!/$.

The *minimization* of tree pattern Q is to find an equivalent tree pattern Q' ($\equiv Q$) of the smallest size [24]. Existing techniques are applicable to our proposed approach. It may impact the efficiency but not the effectiveness of our approach. In the following discussion, we assume that all tree patterns have been minimized.

III. View Filtering with VFILTER

In this section, we study the problem of view filtering. If a query Q is not contained in a view V, then view V cannot be used to answer Q. Although the problem of containment existence between Q and V is coNP-complete, the filtering problem can be efficiently performed. The candidate view set ($\mathcal{V}'$) obtained after filtering is a superset of the set of all views ($\mathcal{V}_Q$) that contain Q. As shown in our experiment, the size of $\mathcal{V}'$ is close to the size of $\mathcal{V}_Q$ (i.e., $\mathcal{V}'/\mathcal{V}_Q \approx 1$). Next we study containment of tree patterns, followed by the discussion of view filtering.

A. Tree Pattern Containment

We define a *decomposition* of a tree pattern Q, denoted by $D(Q)$, as a set of path patterns, each of which corresponds to a root-to-leaf path in Q. Note that $D(Q)$ does not contain duplicate path patterns. For example, given a tree pattern $Q_e = b[.// * [f]/t]// * /t$, we have $D(Q_e) = \{b// * /f, b// * /t\}$. We denote by $|D(Q)|$ the cardinality of the set $D(Q)$, e.g., $|D(Q_e)| = 2$.

Determining pattern containment by homomorphisms is sound but incomplete, since for two patterns Q, Q' where $Q \sqsubseteq Q'$, the homomorphism may not exist from Q' to Q [15]. However, we have the following properties.

Theorem 3.1: *[15] Homomorphism for checking containment $Q \sqsubseteq Q'$ is complete when Q' is a path pattern.* □

Proposition 3.1: *Given two tree patterns Q and Q', the necessary condition for $Q \sqsubseteq Q'$ is that for each path pattern $P' \in D(Q')$, there exists a path pattern $P \in D(Q)$ where $P \sqsubseteq P'$.* □

Proof: (Proof by Contradiction.) Assume to the contrary that for some path pattern $P' \in D(Q')$, there does not exist a path pattern $P \in D(Q)$ where $P \sqsubseteq P'$. We have that Q is not contained in P' based on Theorem 3.1, since no homomorphism exists from P' to Q and P' is a path pattern. Therefore, there exists some XML database $\mathcal{D}_1$ where $Q(\mathcal{D}_1)$ is true while $P'(\mathcal{D}_1)$ is false.

On the other hand, we have $Q' \sqsubseteq P'$ since $P' \in D(Q')$ and a homomorphism exists from P' to itself on Q' (i.e., $Q'(\mathcal{D})$ implies $P'(\mathcal{D})$). Therefore, we have $Q'(\mathcal{D}_1) =$ false since $P'(\mathcal{D}_1)$ is false. The above result contradicts the condition, $Q \sqsubseteq Q'$, since there exists an XML database $\mathcal{D}_1$ where $Q(\mathcal{D}_1)$ is true while $Q'(\mathcal{D}_1)$ is false. ■

Proposition 3.1 states that for a query Q and a view V, if there exists a path pattern in $D(Q)$ that is not contained in any path pattern in $D(V)$, Q is not contained in V. Based on the above property, we build an NFA for view filtering.

B. VFILTER: An NFA-based Model

In this section, we find the candidate views by filtering out the non-candidate ones using an NFA. However, our filtering approach may over-filter some views. We propose a normalization process to amend this in Section III-C.

The NFA-based model that we study is different from previous work (e.g., [25], [26]). Their systems focus on

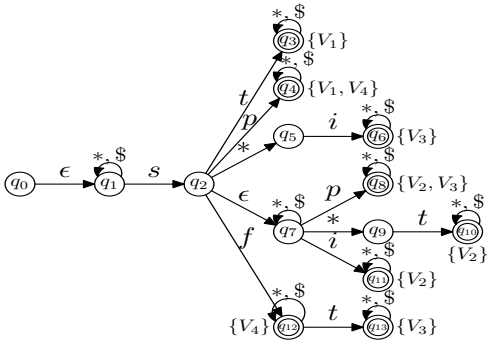

Fig. 4. VFILTER

View	Tree Pattern	|D(V)|
V_1	s[t]/p	2
V_2	s[.//*/t][.//i]//p	3
V_3	s[*/i][f/t]//p	3
V_4	s[p]/f	2

TABLE I
SAMPLE VIEWS

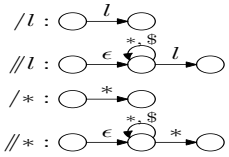

Fig. 5. NFA Fragments

Path	Path Pattern	View
P_1	s/t	V_1
P_2	s/p	V_1, V_4
P_3	s/*/i	V_3
P_4	s//p	V_2, V_3
P_5	s//*/t	V_2
P_6	s//i	V_2
P_7	s/f	V_4
P_8	s/f/t	V_3

TABLE II
PATH PATTERNS

finding the *embeddings* from path patterns to XML documents. Our approach, however, captures the *containment* semantics between path patterns, which is more complicated.

The NFA-based model proposed, called VFILTER, is represented as a *5-ary tuple*: $\mathcal{A} = (\mathsf{S}, \Sigma, \hat{\delta}, \mathsf{q}_0, \mathsf{F})$, where $\mathcal{A}$ is the name of NFA, S is its set of states, Σ its input symbols, $\hat{\delta}$ its transition function, q_0 its start state, and F is the set of accepting states.

VFILTER is built based on the decomposed path patterns of all views. We use \$ to represent the $//$-axis, and a finite set of alphabets of VFILTER is $\Sigma = \mathsf{L} \cup \{*, \$\}$. Note that the $*$, in XPATH semantics, is the label wildcard that matches any label but not query axis (i.e., $*$ matches $\Sigma - \{\$\}$). The \$ (for $//$-axis) can only match \$ but not any label.

The four basic NFA fragments for path patterns are $/l$, $//l$, $/*$ and $//*$, where $l \in \mathsf{L}$. Figure 5 shows the correspondence between path patterns and their NFA fragments. The ϵ-transition is used for combining the basic NFA fragments. Note that the self-loop labeled $*$ and \$ of NFA fragments (e.g., $//l$ in Figure 5) accepts any label and edge, which is in accordance with the $//$-axis semantics of pattern containment (i.e., $//l$ *contains* any path pattern ended with label l).

The automaton of a path pattern is constructed by concatenating its NFA fragments, turning the sink node to the accepting state and adding a self-loop that accepts any label or edge. The automaton $\mathcal{A}$ for multiple path patterns is constructed by combining their common prefixes.

Example 3.1: *Consider a view set $\mathcal{V} = \{V_1, \cdots, V_4\}$ listed in Table I. The decomposition $D(V)$ is the set of distinct path patterns of V. Table II shows all path patterns decomposed from $\mathcal{V}$. The automaton $\mathcal{A}$ constructed using the path patterns shown in Table II is given in Figure 4. All accepting states are highlighted with double-circles.*

The transformation of an input path pattern P to a string w, denoted as $w = \text{STR}(P)$, is described as follows: omit $/$ and replace $//$ by \$. Consider a path pattern $P_e = /b// * /f$, we have $w_e = \text{STR}(P_e) = b\$ * f$. Note that the queries as absolute queries starting with $//$, thus we have $\text{STR}(P'_e) = \$bs$ for $P'_e = b/s$, while not bs, due to XPATH semantics.

Each accepting state f corresponds to a path pattern P_f. If the string $\text{STR}(P)$, for a path pattern P, is in the *language* of $\mathcal{A}$ and f is in an accepting state after reading the string representation for P (i.e., $\text{STR}(P)$), we have $P \sqsubseteq P_f$ since there is a homomorphism from P_f to P. In Figure 4, instead of maintaining path patterns at each accepting state, we maintain corresponding views. For example, the views maintained at state q_4 are $\{V_1, V_4\}$, which means that the path pattern for q_4 (i.e., s/p) can be decomposed from V_1 or V_4.

C. False Negatives Elimination

Next we propose to eliminate false negatives by path pattern normalization, since VFILTER may over-filter views that contain a query unless path patterns are normalized. We then describe the algorithm for view filtering.

Example 3.2: *Consider a path pattern $P_e : s/*//t$, and a transformed string $w_e = \text{STR}(P_e) = \$s * \$t$. Recall that $\widehat{\delta}(\mathsf{q}_0, w_e)$ represents the set of states reached by automaton after reading w_e. We have $\widehat{\delta}(\mathsf{q}_0, \$)$=$\{\mathsf{q}_1\}$, $\widehat{\delta}(\mathsf{q}_0, \$s)$=$\{\mathsf{q}_1, \mathsf{q}_2\}$, $\widehat{\delta}(\mathsf{q}_0, \$s*) = \{\mathsf{q}_1, \mathsf{q}_5, \mathsf{q}_7, \mathsf{q}_9\}$, $\widehat{\delta}(\mathsf{q}_0, \$s * \$) = \{\mathsf{q}_1, \mathsf{q}_7\}$ and $\widehat{\delta}(\mathsf{q}_0, \$s * \$t)$=$\{\mathsf{q}_1, \mathsf{q}_7, \mathsf{q}_9\}$. The automaton $\mathcal{A}$ does not accept w_e since $\{\mathsf{q}_1, \mathsf{q}_7, \mathsf{q}_9\} \cap \mathsf{F} = \emptyset$ (i.e., $\mathsf{q}_1, \mathsf{q}_7$ and q_9 are not accepting states). We have that P_e is not contained by any path pattern maintained in $\mathcal{A}$. Nevertheless, the path pattern P_e is equivalent to P_5 in Table II (i.e., $P_e \equiv P_5$), which should be accepted by $\mathcal{A}$.*

Path pattern normalization has been introduced as a query rewriting technique for normalizing equivalent path patterns [14], [18]. Our normalization process, as described below, is different in that it aims at eliminating false negatives of path pattern matching using VFILTER.

Next we address *path pattern normalization* for eliminating false negatives. Given a path pattern P of the form $\alpha_0 l_0 \alpha_1 * \alpha_2 * \cdots \alpha_n * \alpha_{n+1} l_{n+1}$ where $\alpha_i \in \{/, //\}$, $*$ is the wildcard, $l_0 \neq *$ and $l_{n+1} \neq *$, assume that the number of $//$-edges from α_1 to α_{n+1} is j where $j \leqslant n+1$. If $j = 0$, pattern P is already normalized; otherwise, we normalize P to P' : $\alpha_0 l_0 // * /* \cdots /* /l_{n+1}$ with one $//$-edge only (for α_1) and the number of $/$-edges is n (from α_2 to α_{n+1}). We have $P \equiv P'$ since they have the same result, which means that at least n labels exist between l_0 and l_{n+1} (in case $j > 0$). We say a path

pattern P is normalized, denoted as $N(P)$, if all subsequences containing the label $*$ only are normalized, as stated above. In fact, there are many ways to normalize a path pattern by varying the number of $/\!/$-edges and their positions to get an equivalent pattern (e.g., [14], [18]). We push the $/\!/$-edge to the front for early pruning in VFILTER.

Example 3.3: *Consider the path pattern $P_e : s/*/\!/t$. Although it is equivalent to P_5 in Table II, it cannot be accepted by automaton $\mathcal{A}$. This causes a false negative. By normalizing the path pattern P_e to $N(P_e) = s/\!/*/t$ as indicated in Example 3.2, the pattern is now the same as P_5 that can be accepted by automaton $\mathcal{A}$, and false negatives are eliminated. It is what is being shown in Figure 4.*

The path pattern P and the normalized pattern $N(P)$ are equivalent, which can be proven by combining adjacent $/\!/$-edges and $*$ into a single unit (e.g., [3]). In the above example, both $s/\!/*/t$ and $s/*/\!/t$ can be transformed to $s/\!/^1t$, where character m in $/\!/^m$ represents the number of $/\!/$ for consecutive wildcards $*$. Thus, we have the following:

Proposition 3.2: *If two path patterns are equivalent, then their normalized patterns are the same.* □

Thus, false negatives are eliminated by normalizing path patterns when building VFILTER and reading inputs. Note that all path patterns in Table II have been normalized.

D. View Filtering using VFILTER

In this section, we present an algorithm for view filtering. This algorithm allows false positives but not false negatives, as discussed above. That is, the algorithm may pick a view $V_i \in \mathcal{V}$ that may not satisfy the condition $Q \sqsubseteq V_i$, but it never filters a view $V_j \in \mathcal{V}$ where $Q \sqsubseteq V_j$.

The algorithm VIEWFILTERING() is given in Algorithm 1, with three inputs: a query Q, a view set $\mathcal{V}$, and a VFILTER $\mathcal{A}$. Initially, the number of path patterns of a view V (i.e., NUM(V)) that may accept a path pattern of Q is set to 0 (line 1). Then we decompose Q into a set of path patterns $D(Q)$ and normalize them (lines 2-5). Next, we transform each path pattern in $D(Q)$ to a string and use VFILTER to read it (lines 7-8). We increment NUM(V) each time a path pattern of V accepts some path pattern of Q (lines 11-12). Finally, we output V if each path pattern of V contains some path pattern of Q (lines 17-20).

Algorithm 1 also maintains an auxiliary structure (a sorted list) for each path pattern of query Q (line 9). These lists are used for multiple-view selection (see Section IV). The list of a $P_i \in D(Q)$ (i.e., LIST(P_i)) records a set of pairs (V, l) sorted by l in descending order. Here, V is a view that is contained in P_i (i.e., $V \sqsubseteq P_i$) and l is the length of P (i.e., the number of labels in P). We update LIST(P_i) when P_i is contained in a path pattern of V (line 13); if V is not yet in LIST(P_i), (V, l) is inserted into LIST(P_i); otherwise, l' is set to l if $l > l'$ (we record the largest length only). Finally, we remove the maintained view information from the sorted lists if the views are filtered (lines 22-26).

Algorithm 1 VIEWFILTERING $(Q, \mathcal{V}, \mathcal{A})$

```
1: NUM(V) ← 0 for each V ∈ 𝒱;
2: Decompose Q, D(Q) = {P_1, ..., P_k};
3: for each P_i ∈ D(Q) do
4:    Normalize P_i as P_i ← N(P_i);
5: end for
6: for each P_i ∈ D(Q) do
7:    Transform P_i to a string w_i, w_i ← STR(P_i);
8:    Let R ← δ̂(q_0, w_i) ∩ F;
9:    LIST(P_i) ← NULL;
10:   for each state q ∈ R do
11:      if view V is maintained at state q then
12:         NUM(V) ← NUM(V) + 1;
13:         update LIST(P_i) by a pair (V, l);
14:      end if
15:   end for
16: end for
17: for each V ∈ 𝒱 do
18:   if NUM(V) = |D(V)| then
19:      output V as a candidate view;
20:   end if
21: end for
22: for each P_i ∈ D(Q) do
23:   for each (V, l) pair in LIST(P_i) do
24:      remove (V, l) if NUM(V) ≠ |D(V)|;
25:   end for
26: end for
```

Example 3.4: *Consider a query $Q_e : s[f/\!/i][t]/p$. We decompose Q_e to three path patterns $s/f/\!/i$, s/t and s/p, with transformed strings $w_1(\$sf\$i)$, $w_2(\$st)$ and $w_3(\$sp)$, respectively. Using* VFILTER *in Figure 4 to read w_1, we have $\widehat{\delta}(q_0, \$) = \{q_1\}$, $\widehat{\delta}(q_0, \$s) = \{q_1, q_2\}$, $\widehat{\delta}(q_0, \$sf) = \{q_1, q_5, q_7, q_9, q_{12}\}$, $\widehat{\delta}(q_0, \$sf\$) = \{q_1, q_7, q_{12}\}$ and $\widehat{\delta}(q_0, \$sf\$i) = \{q_1, q_7, q_{11}, q_{12}\}$. We reach the accepting states q_{11} and q_{12}, and NUM(V_2) and NUM(V_4) will be incremented by 1, so that NUM(V_2) = 1 and NUM(V_4) = 1. Reading w_2, we have $\widehat{\delta}(q_0, \$st) = \{q_1, q_3, q_5, q_7, q_9\}$. We reach the accepting state q_3, and NUM(V_1) will be incremented by 1, so that NUM(V_1) = 1. Reading w_3, we have $\widehat{\delta}(q_0, \$sp) = \{q_1, q_4, q_5, q_7, q_8, q_9\}$. We reach the accepting states q_4 and q_8, and all NUM(V_i) $(1 \leqslant i \leqslant 4)$ will be incremented by 1, so that NUM(V_1) = 2, NUM(V_2) = 2, NUM(V_3) = 1 and NUM(V_4) = 2. V_1 and V_4 are selected as candidate views, since NUM(V_1) = $D(V_1)$ and NUM(V_4) = $D(V_4)$, based on Proposition 3.1 and Table I.*

Moreover, the sorted lists maintained for the three path patterns of Q_e are the followings: $\{(V_4, 2)\}$ for $s/f/\!/i$, $\{(V_1, 2)\}$ for s/t, and $\{(V_1, 2), (V_4, 2)\}$ for s/p.

The benefits of using NFA-based implementation for view filtering are threefold. (1) We use PTIME path pattern containment test for view/query containment filtering. (2) The space complexity of VFILTER is $O(mn)$ where m is the number of views and n is the size of the largest view (i.e., the number

of nodes). The NFA-based implementation has a tremendous reduction in path states by sharing their common path prefixes. (3) As pointed out in [26], NFA has low time complexity in inserting/deleting states/transitions, and performing the transitions using a hash table.

IV. Multiple-View Selection

The problems we study in this section are the followings: given a view set $\mathcal{V}'$ and a query Q, where Q is contained in each view of $\mathcal{V}'$, what is the criterion for Q to be answerable by view set $\mathcal{V}'$, and how to select $\mathcal{V}'$ from a candidate view set $\mathcal{V}$. Although finding containment by computing homomorphisms is sound but incomplete, it is rare to find where containment holds but no homomorphism exists. In this paper, we check containment using homomorphisms. Our concern is to discover the potential connection between multiple views to optimize a query.

A. Query Answerability Criterion

We introduce *leaf-cover* for a view V and a query Q, denoted as $\text{LC}(V, Q)$, where a homomorphism (h) exists from V to Q. The intuition behind leaf-cover is that, if a leaf node set $(\mathcal{N})$ of Q can be covered by V, all ancestors of $\mathcal{N}$ can be identified from V, by either examining the materialized fragments of V or verifying encodings. If a view set $\mathcal{V}'$ covers all leaf nodes of Q, then all predicates for Q can be identified, and Q is answerable by $\mathcal{V}'$. The materialized fragments of different views are associated by joins.

We define $\text{LF}(Q) = \text{LEAF}(Q) \cup \{\Delta\}$, where $\text{LEAF}(Q)$ is the set of all leaf nodes of a tree pattern Q, and Δ is a special character for the answer node. The *leaf-cover*, $\text{LC}(V, Q)$, is defined as follows:

1) $\Delta \in \text{LC}(V, Q)$ if the mapped answer node of V is an ancestor-or-self node of the answer node of Q;
2) a leaf node n of Q is in $\text{LC}(V, Q)$, if n is a descendant-or-self node of the mapped answer node of V, or the predicates for n and its ancestors hold on V;
3) $\text{LC}(V, Q) = \text{LF}(Q)$ iff condition 1 holds, and the predicates of Q that are not under $h(\text{RET}(V))$ (i.e., the node of V mapped by Q's answer node) hold on V.

In condition 1, Δ means that the query result is contained in and may be derived from the view result. In condition 2, a leaf node in the leaf-cover means that this node and all its ancestors either can be identified from the view result or hold on view. Condition 3 states that a single view can answer a query, if the query result can be identified from the view result, and all predicates either can be derived from the view result or are satisfied by the view via a homomorphism. Note that $\text{LC}(V, Q) = \emptyset$ if Q is not contained in V.

Consider four materialized XPATH views in Figure 6(a-d) and two XPATH queries in Figure 6(e-f), where answer nodes are circled. We have the following:

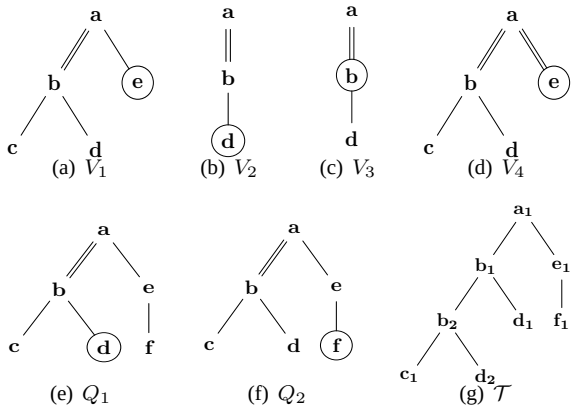

Fig. 6. Sample views, queries and XML tree (a) $V_1 : a[.//b[c]/d]/e$ (b) $V_2 : a//b/d$ (c) $V_3 : a//b[d]$ (d) $V_4 : a[.//b[c]/d]//e$ (e) $Q_1 : a[e/f]//b[c]/d$ (f) $Q_2 : a[.//b[c]/d]/e/f$ (g) an XML tree $\mathcal{T}$

$$\begin{array}{ll} \text{LC}(V_1, Q_1) = \{f\} & \text{LC}(V_1, Q_2) = \{\Delta, c, d, f\} \\ \text{LC}(V_2, Q_1) = \{\Delta, d\} & \text{LC}(V_2, Q_2) = \{d\} \\ \text{LC}(V_3, Q_1) = \{\Delta, c, d\} & \text{LC}(V_3, Q_2) = \{c, d\} \\ \text{LC}(V_4, Q_1) = \{f\} & \text{LC}(V_4, Q_2) = \{\Delta, f\} \end{array} \quad (1)$$

Criterion for answerability: A view set $\mathcal{V}$ can answer a query Q if $\bigcup_{V \in \mathcal{V}} \text{LC}(V, Q) = \text{LF}(Q)$.

Above criterion states that if a group of views can answer a query, all predicates of query leaves and their ancestors (i.e., the whole query) either hold through homomorphism, or can be identified from view results. $\mathcal{V} \vDash Q$ denotes that a view set $\mathcal{V}$ can answer a query Q. Based on this criterion and Equation 1, we can easily derive the cases listed in Equation 2. Here, $\text{LF}(Q_1) = \{\Delta, c, d, f\}$ and $\text{LF}(Q_2) = \{\Delta, c, d, f\}$.

$$\begin{array}{lll} (V_1, V_2) \nvDash Q_1 & (V_1, V_3) \vDash Q_1 & (V_1) \vDash Q_2 \\ (V_2, V_4) \nvDash Q_1 & (V_3, V_4) \vDash Q_1 & (V_3, V_4) \vDash Q_2 \end{array} \quad (2)$$

Example 4.1: *Consider the views and queries in Figure 6. Each query (Q_1 or Q_2) is contained in every one of the views (V_1 to V_4). However, no single view can be used to answer Q_1. The result of Q_1 is not contained in the materialized fragments of V_1 and V_4; V_2 and V_3 cannot be utilized even if their results contain Q_1's result. Consider V_3 for example. We cannot tell which materialized b-nodes have an ancestor a that satisfies the predicate $[e/f]$; the reason for V_2 is similar. Only V_1 can be used to answer Q_2. V_4 cannot answer Q_2, since we cannot identify which materialized e-nodes of V_4 have a parent a that satisfies $[.//b[c]/d]$. The result of Q_2 is not contained in the results of V_2 and V_3 and cannot be answered by V_2 or V_3.*

Consider, however, V_1, V_3 and Q_1. Assume that we can identify the materialized e-nodes of V_1 that have child f, and b-nodes of V_3 that have children c and d. If the identified b-nodes of V_3 have ancestors a that are, in turn, parents of those identified e-nodes of V_1, we can extract Q_1's result from the

refined materialized fragments of V_3 since all predicates of Q_1 hold (i.e., $(V_1, V_3) \vDash Q_1$).

Next we reason by example why $(V_1, V_2) \nvDash Q_1$, although it appears to hold.

Example 4.2: *Consider V_1, V_2 and Q_1 in Figure 6. Intuitively, we can identify the e-nodes of V_1 that have child f, then join with the d-nodes of V_2 to derive Q_1's result. However, consider the* XML *tree in Figure 6(g). d_1, d_2 are the results of V_2 and e_1 is the result of V_1 that has child f_1. If we join d_1, d_2 with e_1 that have a common ancestor a, which is the parent of e_1 and an ancestor of some b-node that is, in turn, the parent of d, we obtain d_1 and d_2. But indeed d_1 is not, since we cannot distinguish which d-node has a parent b that, in turn, has child c.*

B. Selecting Multiple Views

Let Q be a query that is answerable by a view set $\mathcal{V}$. We say $\mathcal{V}$ is a *minimum rewriting* (for global minimum) if there is no other view set $\mathcal{V}'$ that answers Q with fewer views than $\mathcal{V}$. We say $\mathcal{V}$ is a *minimal rewriting* (for local minimum) if no subset $\mathcal{V}'' \subseteq \mathcal{V}$ answers Q. In this section, we describe a naive method to find the minimum rewriting, and a heuristic method to compute a minimal rewriting. Homomorphisms between the views and the query are used for rewriting (see Section V). Next we state multiple-view selection.

Given a view set $\mathcal{V}$ and a query Q, find a subset $\mathcal{V}' \subseteq \mathcal{V}$ with the least number of views, such that $\bigcup_{V \in \mathcal{V}'} \text{LC}(V, Q) = \text{LF}(Q)$ (i.e., $\mathcal{V}' \vDash Q$). By computing all leaf-covers from each $V \in \mathcal{V}$ to Q, the classical NP-hard *set covering* problem can be reduced to this problem.

Finding a minimum rewriting. A naive method to find the minimum rewriting requires computing all possible combinations of sets, and outputting the one with the least number of views. This is $O(2^{|\mathcal{V}|})$ in the worst case.

A heuristic method for finding a minimal rewriting. We propose a heuristic method to avoid computing homomorphisms of all views to the query, which is expensive when the number of views is large. Given a view set $\mathcal{V}'$, a query Q where $D(Q) = \{P_1, \cdots, P_k\}$, and the sorted lists (i.e., $\text{LIST}(P_i)$ in Section III-D) maintained for each path pattern $P_i \in D(Q)$, multiple views are selected so that all query leaf nodes are covered by the views and the other nodes of Q can be identified from the views. Moreover, the result of Q can be extracted from the result of some view (as discussed in Section IV-A). The heuristic method requires scanning each view once in the worst case, which is $O(|\mathcal{V}|)$.

The minimum view set focuses on finding the smallest view set. The heuristic method, however, aims at finding the views with the smallest view fragments. Our experiments show that query performance from the heuristic method is better than the method with minimum view set. A cost model that combines above two factors (i.e., the number of views and the size of view fragments) may achieve better performance. We omit this discussion due to space limitation.

Algorithm 2 VIEWSELECTION $(Q, \mathcal{V})$

```
1:  V' = ∅;
2:  while leaf node set LF(Q) ≠ ∅; do
3:     randomly select a node u ∈ LF(Q);
4:     take P ∈ D(Q) that u belongs to;
5:     repeat
6:        select one pair (V, l) from LIST(P);
7:        compute LC(V, Q) by a homomorphism;
8:        if u ∈ LC(V, Q) then
9:           flag ← true
10:          V' ← V' ∪ {V};
11:          LF(Q) ← LF(Q) − LC(V, Q);
12:       else
13:          select the next pair (V', l') from LIST(P)
14:       end if
15:    until flag = true or the end of LIST(P)
16:    if flag = false then
17:       return "the query Q is not answerable by V";
18:    end if
19: end while
20: V' ← V' − {V} if LC(V, Q) ⊆ ⋃_{V'∈V'−{V}} LC(V', Q)
21: return V';
```

We describe the heuristic multiple-view selection in Algorithm 2. Initially, no view is selected (line 1). We randomly select a leaf node u of query Q that is not yet covered by any view (line 3). Next we take one view from the maintained list of P (line 6), and compute the homomorphism from V to Q (line 7). If the leaf node u is covered, we add V to the selected view set $\mathcal{V}'$, and remove all leaf nodes covered by V from $\text{LF}(Q)$(lines 8-11). Recall that the list of P is sorted by length in descending order, where the length records the longest path of each view that contains P. This algorithm greedily uses a view that is as large in length as possible, which means that the compensating query will be executed on a smaller view result. The view is not answerable if some leaf node is not covered by any view (lines 15-18). Finally, when all the leaf nodes are covered (from the loop in lines 2-19), we remove redundant views and output the view set $(\mathcal{V}')$ that can answer Q (lines 20-21). Note that this heuristic algorithm always finds a minimal set.

Example 4.3: *Consider the last example in previous section where we showed how it works on query $Q_e : s[f//i][t]/p$. Here, $\text{LF}(Q_e) = \{\Delta, i, p, t\}$, the candidate views $\mathcal{V}' = \{V_1, V_4\}$ are given in Table I. The lists for path patterns of Q_e are the following: $\{(V_4, 2)\}$ for $P_1' : s/f//i$, $\{(V_1, 2)\}$ for $P_2' : s/t$, and $\{(V_1, 2), (V_4, 2)\}$ for $P_3' : s/p$.*

Next we show how to select multiple views. We first select one leaf node from $\text{LF}(Q_e)$, e.g., i, and the path pattern P_1' in which i is contained. We select the first view from $\text{LIST}(P_1')$ (i.e., V_4) and compute the homomorphism from $V_4 : s[p]/f$ to Q_e. We have $\text{LC}(V_4, Q_e) = \{i, p\}$. Next we remove the covered leaf nodes from $\text{LF}(Q_e)$ and get $\text{LF}(Q_e) = \{\Delta, t\}$. Then we select P_2' in which leaf t is contained. We select view

V_1:$s[t]/p$ *from* LIST(P_2') *and compute* LC(V_1, Q_e)=$\{\Delta, t, p\}$. *Leaf node* t *is covered, and the result node of* Q_e *(i.e.,* p*) can be mapped from the result node of* V_1 *(i.e.,* p*). Algorithm 2 returns* $\{V_1, V_4\}$.

V. Rewriting Using Multiple Views

This section describes how to find an equivalent rewriting using multiple views. Different from the approaches on single-view rewriting that find a compensating query only, we find a compensating query for each materialized view and then join the refined materialized view fragments. Also different from the naive method that combines all materialized view fragments first and then treats it as a single view for rewriting the query, our approach applies the compensating query for each view first (like pushing selection) as an optimization, and then performs a holistic twig join on the roots of all refined view fragments.

Figure 7 shows a view set $\mathcal{V}$ and a query Q that is answerable by $\mathcal{V}$. Without loss of generality, we assume that the answer node of V_1 maps to some ancestor-or-self node of the answer node of Q. We first refine each materialized fragment of $V \in \{V_2, \cdots, V_n\}$ by applying the predicates that are not satisfied in Q, to guarantee that the materialized fragments satisfy these predicates. There are two cases to consider. The first is that the tree pattern rooted at RET(V) is equivalent to the tree pattern rooted at the mapped node h(RET(V)) (e.g., V_2), and V does not need to be refined. The second case is when the predicates for the query are more restrictive than those of the view, e.g., V_n. We need to refine the result for r_n by applying the predicates satisfied on y but not on r_n. The other views are handled similarly.

When views $\{V_2, \cdots, V_n\}$ are refined, we utilize a holistic twig join algorithm on all the answer nodes of views (i.e., $\{r_1, \cdots, r_n\}$) to refine the answer node of V_1 (i.e., r_1) to ensure that all predicates except those under the mapped node of r_1 (i.e., node a in Q) hold. The holistic join algorithm is similar to `TJFast` that uses `extended Dewey-code` [22]. Finally, we apply a compensating query on the refined result of r_1 to extract the result of Q. Note that we use a holistic join algorithm, which requires only one scan of all roots of fragments and runs in linear time. The complete pseudo-code for the join algorithm is not hard but long and complicated. Due to space limitation, we omit it and only give an illustrating example.

Example 5.1: *Consider the database in Figure 2 and two views* $V_1 : s[t]/p$, $V_2 : s[p]/f$ *that may answer a query* $Q_e : s[f/\!/i][t]/p$. *The materialized fragments for* V_1 *are rooted at* $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$, *and for* V_2 *they are rooted at* $\{f_1, f_2, f_3\}$. *We first refine each materialized* f *node of* V_2 *for predicate* $[./\!/i]$, *and no node is filtered. Then we join all these* p *nodes and* f *nodes with a common parent* s *using their encodings as given in Figure 2. The parents of* p_1 (`0.8.1`) *and* f_1 (`0.8.6.3`) *are* `0.8` *and* `0.8.6`, *respectively.* p_1 *is not a result, since their parents are not identical. The case for* p_2 *is similar and thus it is filtered. Next, consider* p_3 (`0.8.6.1`), *whose parent* (`0.8.6`) *is identical to the parent of* f_1. *Based on the* FST *in Figure 3, we can derive the label-path for* `0.8.6` *as* $b/s/s$ *(see Example 2.1).* p_3 *is a result since the common parent* `0.8.6` *of* p_3 *and* f_1 *is labeled* s. *Similarly, we can derive the other results:* $\{p_4, p_5, p_6, p_7\}$.

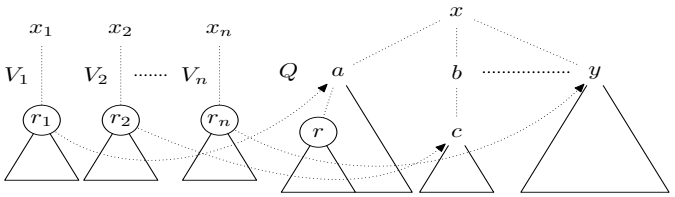

Fig. 7. Views $\mathcal{V}$ and a query Q

Handling comparison predicates. Our approach can be easily extended to handle comparison predicates (i.e., predicates for attributes). The only condition is that the attribute predicates of the query must be exactly the same as those of corresponding mapped view nodes, or can be evaluated from the view results, since the attribute predicates can not be handled upon the encodings.

VI. Performance Study

We conduct two sets of performance tests. Firstly we evaluate the performance of answering queries by multiple views, as well as the actual lookup time of different selection strategies with or without VFILTER. Secondly we evaluate the efficiency of VFILTER, including its size, filtering time, and filtering impact.

The experiments were conducted on a dual `2.0GHz` machine with `1GB` of memory, running `Windows XP`. The algorithms were coded in `C++`. We used `Berkeley DB`[2] and `Berkeley DB XML`[3] for storing VFILTER and XML fragments, respectively. We used `YFilter`[4] as the XPATH query generator, which can generate XPATH queries with assigned the maximum depth (max_depth), probabilities of wildcards $*$ (prob_wild) and descendant edges $/\!/$ (prob_dedge), the number of predicates (num_pred) and the number of nested paths (num_nestedpath). We wrote a program to find *positive* queries (i.e., the result is not empty). We used a `56.2MB` XML document generated by `XMark`[5] generator. We set the size limit for the materialized fragments of each view to `128KB` (the same as [19]), since the query performance on large un-indexed fragments may be worse than that on the well indexed XML database directly.

A. Query Processing Performance

For the workloads we generated for our experiments, the `max_depth` is set to `4`, the `prob_wild` and `prob_dedge` are both set to `0.2`, the `num_pred` is set to `1` and the `num_nestedpath` is set to `1`. We generate and select `1000`

[2]http://www.oracle.com/database/berkeley-db.html
[3]http://www.oracle.com/database/berkeley-db/xml/index.html
[4]http://yfilter.cs.umass.edu
[5]http://monetdb.cwi.nl/xml

Q_1 :/site/people/person[@id="person0"]/name
Q_2 :/open_auction[@id="open_auction10"] [initial][quantity]//bidder/date
Q_3 :/site/regions/samerica/item[.//quantity][name] [@featured="yes"][@id="item10385"]/location
Q_4 :/site/regions/africa/item[@id="item0"] [incategory[@category="category393"]]/name

TABLE III
TESTING QUERIES

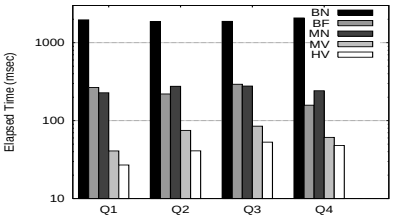

Fig. 8. Processing Time

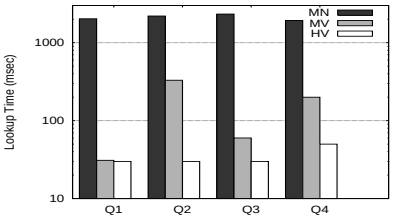

Fig. 9. Lookup Time

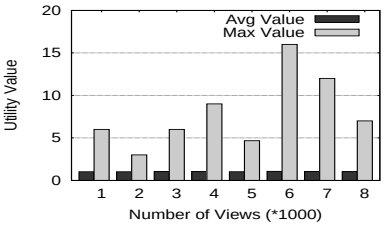

Fig. 10. Utility Ratio

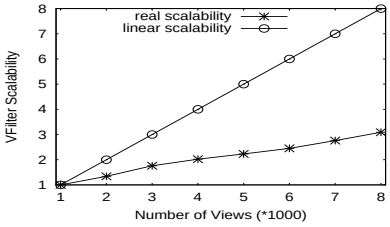

Fig. 11. VFILTER Scalability

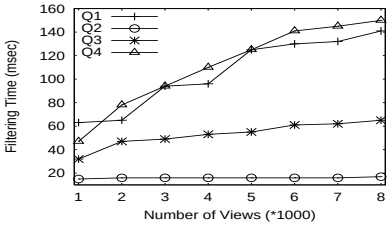

Fig. 12. Filtering Time

positive queries for materialization. The four test queries extracted based on the XMark project are listed in Table III. Q_1 is answered by one view, Q_2 and Q_3 are answered by two views and Q_4 is answered by three views. Note that y-axes in Figure 8 and 9 are logarithmic.

Figure 8 reports the query processing time of different approaches. BN refers to executing queries directly on the XML database with basic node index support. BF refers to executing queries directly on the XML database with full index support to accelerate query performance. MN and MV correspond to answering the query with the minimum set of views using multiple views without/with VFILTER, respectively. HV is the heuristic algorithm with VFILTER.

From Figure 8 we can see that executing the query on the XML database with only node index (BN) is very slow, while it is much faster with full index support (BF). However, the full index support for each node has very high space cost. For the 56.2MB XMark document, the database size with basic node index is 150MB, but the size with full index support is 635MB. The processing time of MN is even slower than running the query on base data directly (e.g., BF). The reason is that the time used for computing homomorphisms is high when the number of views is large. The time complexity for computing the homomorphism from a query Q to a view V is $O(|Q|^2||V|)$ where $|Q|$ is the number of nodes in the tree pattern Q. Note however that we can achieve a comparable query performance on a much smaller XML fragment (128KB limitation per view). With VFILTER support, the algorithms MV and HV achieve significant speedup. However, the heuristic method HV is always better than MV, since HV always greedily finds the views that have smaller materialized fragments while MV only tries to find the least number of views, whose materialized fragments may be larger.

Lookup performance. Next we study lookup performance for a query Q returning a view set $\mathcal{V}$ that answers Q. Figure 9 reports the lookup time for the four test queries on 1000 materialized views. We can see that without the support of VFILTER, algorithm MN always requires considerable time to find the homomorphism of each view to the query, and then find a minimum view set. It is obvious that the lookup time of MN will increase exponentially as the number of views increases, since finding a minimum set of views (e.g., MN) is a NP-hard problem. With VFILTER to filter some views, however, both MV and HV perform lookup very fast, since the number of candidate views generated by VFILTER is small. Furthermore, the filtering time dominates the lookup time when the number of candidate views is small.

B. Performance on VFILTER

In this part, we study the performance of VFILTER. To perform a general test, we generate large sets of views and queries. We generate 8 test sets, $\mathcal{V}_1$ to $\mathcal{V}_8$, with the number of queries 1000, 2000, ..., 8000. All queries are generated with the following parameters: max_depth is set to 4, prob_wild and prob_dedge are set to 0.2, num_nestedpath is set to 2. Note that we do not generate attribute predicates for this test, since we aim at verifying the efficiency of VFILTER for structural filtering.

The first test is based on a utility function $\mathtt{U}(Q) = |\mathcal{V}'|/|\mathcal{V}_Q|$, where $\mathcal{V}'$ is the view set obtained from VFILTER, and $\mathcal{V}_Q$ is the set of views that have homomorphisms to a given query Q. We have $\mathtt{U}(V) \geqslant 1$ since $\mathcal{V}_Q \subseteq \mathcal{V}'$. We use $\mathcal{V}_1$ as the test query set on the test view sets $\mathcal{V}_1$ to $\mathcal{V}_8$. We report the average utility value and maximum utility value in Figure 10. The result shows that the average utility value is very close to 1 for all views in the test view set. This indicates that the case where each path pattern of V contains some path pattern of Q but Q is not contained in V (i.e., $Q \sqsubseteq V$) is not common. The max utility value in this figure shows that the worst result obtained from VFILTER ranges from 3 to 16. We further examine the candidate view set and find that its size (i.e. $|\mathcal{V}'|$) is not larger than 50. This is a surprising result, since

we can always achieve a small number of candidate views with a small cost of computation. The reason is that false positives of VFILTER come from the fact that distinct tree patterns (e.g., $a[b/d]/c/d$ and $a/b[c]/d$) having the same path patterns (i.e., $a/b/c$ and $a/b/d$) appear rarely in the test queries.

Next we study the size of VFILTER constructed from 8 sets of views, from $\mathcal{V}_1$ (1000 views) to $\mathcal{V}_8$ (8000 views). We show the scaling performance, w.r.t. number of views in Figure 11. We denote by $\mathtt{S}_1$ the database size of VFILTER constructed for view set $\mathcal{V}_1$, $\mathtt{S}_2$ the database size of VFILTER for $\mathcal{V}_2$, and so on. The scaling function is defined as $\mathtt{S}_\mathtt{i}/\mathtt{S}_1$ for $1 \leqslant \mathtt{i} \leqslant 8$, and the size of $\mathtt{S}_1$ is 664KB. The scaling performance is given in Figure 11. Intuitively, the scaling performance will be linear to the increasing number of views used to construct VFILTER. However, the result shows that real scaling performance is much smoother than linear scaling performance. The reason is that, along with the increasing number of views, more path patterns will share common prefixes. Therefore, fewer states will be created when inserting a path pattern into a large VFILTER. Considering $\mathtt{S}_8$ that maintains 8000 views, its size is only 2050KB and $\mathtt{S}_8/\mathtt{S}_1 \approx 3.09$. We may tell that the size of VFILTER will keep compact for a larger set of views.

Lastly we study the filtering time of Q_1 to Q_4 (in Table III) using automata constructed from distinct number of views. We report the performance result in Figure 12. The first observation is that VFILTER is efficient for all queries, with the filtering time ranging from 15 msec to 150 msec. The filtering time of Q_2 is almost constant, since Q_2 has the smallest maximum depth among all queries, which is 3, and fewer states will be reached in all scaled automata. For the other queries, the filtering time increases along with the increasing number of views maintained by VFILTER. However, the trend of increasing filtering time is much smoother than the increasing of number of views. Considering the curve of Q_4 that is the most steep, the filtering time of VFILTER with 1000 views is 47msec, and the one with 8000 views is 150 msec, with the approximate ratio 3.2, which is much smaller than 8 (i.e., $|\mathcal{V}_8|/|\mathcal{V}_1|$).

VII. Conclusion

We described a framework for answering XPATH queries using multiple materialized views. We defined multiple view/query answerability, and, by leveraging extended Dewey-code, we can answer a given query with multiple views without accessing the base data. We showed the correctness of decomposing tree pattern query to path pattern queries for filtering. Based on this decomposition, we proposed an NFA-based approach, called VFILTER, for view filtering. We further proposed a heuristic method to find a minimal set of views that answers a given query Q. The performance study showed that VFILTER was compact and efficient.

We plan to incorporate attributes into VFILTER to gain further pruning power. In addition to the equivalent rewriting using multiple views, we also plan to study on maximal rewriting using multiple views in data integration scenario, and multiple partial materialized views when exploring massive data sets.

Acknowledgment

This work was partially supported by a grant of RGC, Hong Kong SAR, China (No. 418205) and by the CUHK strategic grant (No. 4410001).

References

[1] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases." in *VLDB*, 1997, pp. 436–445.
[2] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries." in *SIGMOD*, 2002, pp. 133–144.
[3] T. Milo and D. Suciu, "Index structures for path expressions." in *ICDT*, 1999, pp. 277–295.
[4] N. Zhang, M. T. Özsu, I. F. Ilyas, and A. Aboulnaga, "FIX: Feature-based indexing technique for XML documents." in *VLDB*, 2006, pp. 259–270.
[5] H. He and J. Yang, "Multiresolution indexing of XML for frequent queries." in *ICDE*, 2004, pp. 683–694.
[6] C. Qun, A. Lim, and K. W. Ong, "D(k)-index: An adaptive structural summary for graph-structured data." in *SIGMOD*, 2003, pp. 134–144.
[7] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching." in *SIGMOD*, 2002, pp. 310–321.
[8] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan, "Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents." in *VLDB*, 2006, pp. 283–294.
[9] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic twig joins on indexed XML documents." in *VLDB*, 2003, pp. 273–284.
[10] A. Y. Halevy, "Answering queries using views: A survey." *VLDB J.*, vol. 10, no. 4, pp. 270–294, 2001.
[11] J. Mylopoulos, S. Schuster, and D. Tsichritzis, *A multi-level relational system*, in Proc. 1975 Nat. Computer Conf, AFIPS Press, Arlington, Va.
[12] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System r: Relational approach to database management." *ACM Trans. Database Syst.*, vol. 1, no. 2, pp. 97–137, 1976.
[13] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of ingres." *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, 1976.
[14] G. Miklau and D. Suciu, "Containment and equivalence for an XPath fragment." in *PODS*, 2002, pp. 65–76.
[15] ——, "Containment and equivalence for a fragment of XPath." *J. ACM*, vol. 51, no. 1, pp. 2–45, 2004.
[16] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh, "A framework for using materialized XPath views in XML query processing." in *VLDB*, 2004, pp. 60–71.
[17] L. V. S. Lakshmanan, H. Wang, and Z. J. Zhao, "Answering tree pattern queries using views." in *VLDB*, 2006, pp. 571–582.
[18] W. Xu and Z. M. Özsoyoglu, "Rewriting XPath queries using materialized views." in *VLDB*, 2005, pp. 121–132.
[19] B. Mandhani and D. Suciu, "Query caching and view selection for XML databases." in *VLDB*, 2005, pp. 469–480.
[20] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, "Structural joins: A primitive for efficient XML query pattern matching." in *ICDE*, 2002, pp. 141–.
[21] P. F. Dietz, "Maintaining order in a linked list." in *STOC*, 1982, pp. 122–127.
[22] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen, "From region encoding to extended dewey: On efficient processing of XML twig pattern matching." in *VLDB*, 2005, pp. 193–204.
[23] X. Wu, M.-L. Lee, and W. Hsu, "A prime number labeling scheme for dynamic ordered XML trees." in *ICDE*, 2004, pp. 66–78.
[24] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava, "Minimization of tree pattern queries." in *SIGMOD*, 2001, pp. 497–508.
[25] M. Altinel and M. J. Franklin, "Efficient filtering of xml documents for selective dissemination of information." in *VLDB*, 2000, pp. 53–64.
[26] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering." *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467–516, 2003.